a validator program which, using ~~the~~ said predefined rules of syntax and semantics of said formal language, verifies that every statement in said formal specification is syntactically complete, semantically correct and not ambiguous.

REMARKS

REMARKS

Claims 1-24 have been rejected as obvious over the combination of Goodwin et al. (US 6,199,195) in view of Features of VDM Tools and further in view of Koob et al.

In response to this rejection, the preamble of claim 1 has been amended to further clarify the difference between the Godwin system and the system claimed in claim 1. The apparatus of claim 1, as amended, is a tool for receiving user input that defines a conceptual model of a complete and operable software program to be automatically generated and converts said conceptual model to a specification for the software written in a formal language such as OASIS which has precise rules of syntax and semantics. The tool then validates the formal language specification using the rules of syntax and semantics of the formal language to ensure that the specification is complete, semantically correct and not ambiguous. The resulting formal language specification may then be input to a translator and be converted into a real, operating and complete computer program in any computer language for which there is a translator available. There is no limitation or requirement that the computer program be object oriented in the claimed invention.

In contrast, the Goodwin et al. system does not create complete formal language specifications for a complete, operable computer program. The Goodwin system outputs software in the form of "objects within an extensible object framework". This means that the software objects generated by the Goodwin system are not a complete, operable computer program nor are they a full specification for such a program written in a formal language. The "extensible framework" is software which is not written by the Goodwin et al. system which is needed to complete the computer program. In other words, the objects output by the Goodwin et al. system are not a complete computer program and cannot work by themselves -- they need the software of the "extensible framework" to be complete and operable. Presumably, the extensible framework is written by hand to embody such things as a framework of services, user interface, etc. This extensible framework is authored by the human software developers using the Goodwin tool and is not generated by the Goodwin tool itself, as evidenced by the bold part of the following passage from Col. 6, lines 37-41:

> "The disclosed system and method allow object developers to design and author new object services, and to define how these services are composed within extensible frameworks with other object services. This is accomplished through object templates, which make up the system definition 208, and describe the object services and their dependencies in relationships to other object services as well as behaviors to be generated for objects within the framework.
>
> The system and method will also allow developers to generate objects **based on a framework of services they author** by composing services based on the object templates into objects that support the composed behaviors and methods."

Thus, it appears that, unlike the claimed invention of claim 1, the Goodwin tool does not generate a formal language specification for a complete, operable computer program -- Goodwin et al. only software objects within the human written "extensible framework" and these objects comprise neither and complete, operable program nor a formal language specification therefor.

The invention of claim 1 outputs a full, validated formal language specification for a complete, operable computer program which does not need to be completed by third party source code, existing components or code libraries. The formal language specification is not limited to specification of a computer program written in an object oriented computer language nor even employ a specific software architecture.

There is ample evidence in Goodwin et al. that Goodwin's tool does not generate a full computer program.

For example, at Col. 13, Lines 65-67:

> "Output from the code generator can be combined with other user defined codes to create an application."

At Col. 13, Lines 4-9:

> "User's own code can be combined with the generated files to produce a resultant code library. The library includes user defined behaviors and support for each of the selected services for the given framework, and for interacting with the particular data model represented in the unified model."

At Col. 17, Lines 10-14:

> "The fourth step is adding the code from the developer that actually implements the methods defined for the unified model, and any custom services or user defined code from the developer. (Business logic is included in this step; this step is a development time step.)

Clearly these passage prove that human developer written code must be supplied along with the objects generated by the Goodwin tool to make a complete program. Further, the Goodwin et al. patent makes no mention of generation of a specification in a formal language from user input and validation of that specfication, as called for by claim 1.

Thus, Goowin et al. does not teach all the knowledge needed to make the claimed invention of claim 1 since it does not teach automatic generation of a full, formal language specification for a complete and operable computer program. That raises the question regarding whether the other two references fill this gap and supply the missing knowledge.

The article "Features of VDMTools" describes a toolbox that supports the development of "formal specifications" using the ISO VDM-SL standard. Page 1, lines 1-2. The question this statement raises is whether the "formal specification" mentioned in the "Features of VDMTools" article is a formal language specification in the sense that term is used in claim 1 and the other claims.

DEFINITION OF FORMAL LANGUAGE SPECIFICATION AND AUTOMATIC VALIDATIONTHEREOF

The Examiner indicated the "formal specification feature is unclear" and indicated he was interpreting the formal specification feature as "the converted portion in Goodwin being validated."

The proper interpretation of the terms "formal language" and "formal specification" and "automatically validating said formal language specification" as these terms are used in the claims at bar can be determined from the specification. These terms are used to define the nature of the specification statements which are generated from the conceptual model built by receiving user input and the process of automatically determining that these formal language statements are "syntactically complete, semantically correct and not ambiguous".

The term "formal language" is to be interpreted as a language which has a grammar defined by rules of syntax and semantics. Syntax defines the correct structure and order of elements in statements in the formal language, and semantics is the meaning of every element in statements in the formal language. More precisely, the semantics of a formal language or symbology that must be used to practice the claimed invention means that everything in the formal language statements of the specification is defined and has one and only one meaning so there can be no ambiguity caused by multiple meanings for the same term.

The fact that a formal language is used to write the specification is proved from the highlighted sections of the following passage from page 17 of the specification at bar:

> Preferably, the CASE tool 210 employs object-oriented modeling techniques to avoid the complexity typically associated with the use of purely textual formal methods. In one implementation, the Conceptual Model is subdivided into four complementary models: an object model, a dynamic model, a functional model, and a presentation model. These models are described in greater detail hereinafter. After gathering the requirements 200, **the CASE tool 210 stores the input requirements as a formal specification 215 in accordance with a formal specification language, for example, the OASIS language,** which is an object-oriented language for information systems developed at developed at the Technical University of Valencia, Spain.<u>Using extended grammar defined by the formal language, the validator 220 syntactically and semantically validates the formal specification 215 to be correct and complete.</u> If the formal specification

215 does not pass validation, no application is allowed to be generated; therefore, only correct and complete applications are allowed be generated.

Further support for the meaning of formal language specification is found in the following passage from page 32 of the specification at bar:

The CASE tool 210, after presenting a user interface for capturing system requirements 200, converts the system requirements into a formal specification 215 in a formal language having a syntax and semantics that are known to the validator 220. Although the formal specification 215 must be in a formal language, it need not be in a known formal language, and any formal language including newly invented formal languages will suffice. The only thing that is necessary to practice the invention is that the syntax and semantics of whatever formal language in which formal specification 215 is written, the validator 220 must know that syntax and semantics so that it may validate the formal specification for completeness, mathematical and semantic and syntactical correctness and lack of ambiguity. In particular the CASE tool 210 builds upon the previously described models as a starting point and automatically generates a corresponding formal and object-oriented specification 215, which acts as a high-level system repository. In a preferred embodiment, the formal language being employed is OASIS, in its version 2.2, published in October 1995 by the "Servicio de Publicaciones de la Universidad Politecnica de Valencia" (legal deposit number: V-1285-1995).

Conversion of captured system requirements 200 into a formal language specification 215 is a main feature of one aspect of the invention: each piece of information introduced in the conceptual modeling step has a corresponding formal language counterpart, which is represented as formal language statements having syntax and semantics known to the validator.

Claim 1 calls for validation of the formal language specification using a validator program. The way the Examiner should intepret this claim limitation is best understood by some specific examples taught in the specification as shown in the following passages from page 33 et seq. of the specification at bar:

As an example of syntax and semantics of formal languages and how the validator 220 can validate such a formal language specification, consider Table 3 below. Table 3 is a formal specification in the OASIS formal language of the reader class of the hypothetical library management application detailed above. TABLE 3 shows a formal specification 215 for the *reader* class that was automatically obtained from the Conceptual Model:

TABLE 3

```
CONCEPTUALSCHEMAlibrary
domains nat,bool,int,date,string

class reader
identification
by_reader_code: (reader_code);
constant_attributes
age : String ;
reader_code : String ;
name : String ;
variable_attributes
```

```
book_count : Int ;
private_events
new_reader()  new;
destroy_reader() destroy;
punish();
shared_events
loan() with book;
return() with book;
constraints
static book_count < 10;
valuation
[loan()] book_count= book_count + 1;
[return()] book_count= book_count - 1;
preconditions
librarian:destroy_reader () if
book_number = 0 ;
triggers
Self :: punish() if book_count = 10;
process
reader = librarian:new_reader() reader0;
reader0= librarian:destroy_reader() +
loan () reader1;
reader1= if book_count=1 return() reader0
+ (if book_count > 1 return()
+ if book_count < 10 loan()) reader1;
```

```
end_class


ENDCONCEPTUALSCHEMA
```

Consider the following statement from the high level repository formal specification of Table 3:

```
[loan()] book_count= book_count + 1;
```

The semantics of this formal language statement indicate by the () that loan is a service which performs the mathematical computation represented by the equation outside the square brackets. This mathematical formula means that the value of the attribute book_count will be incremented by 1 when this service is executed. The formula could be any other formula where one attribute is set equal to the value of another attribute plus the value of some other attribute or user input value. However, to be semantically correct, an integer or floating point number cannot be added to an alphanumeric string or any other type of attribute which has no meaning when attempting to add it to an integer or floating point number.

As another example of validation of the semantics of the formal language specification, when an integer is added to a floating point number, the result must be a floating point number and not an integer.

Another example of validation of the semantics involves verifying that for every attribute that has been defined as a variable, there is a service which changes the value of that attribute. Another example of semantic validation is verifying that for every constant attribute, there is no service which attempts to change its value. Another example of semantic validation is if a service "destroy" erases or eliminates an attribute, it makes no sense to modify the attribute after it no longer exists. The validator would flag as an error any formal specification statement which attempted to do so.

One of the functions of the validator is to check the semantics of every statement to make sure that no mathematical formulas attempt to combine entities that are not mathematically combinable, that combining different types of numbers results in the correct type of output number, that nothing gets divided by zero, and that other operations that are mathematically undefined are not required by the formal specification. Stated another way, one function of the validator is to make sure that every formula is well formed, complete and consistent.

The following passage from page 41 of the specification defines the properties of a formal language:

Formal specification languages benefit from the ability of formal environments to ensure that formal specifications 215 are valid or can be checked to be valid. Formal languages define a grammar that rules language expressiveness.

Two procedures are used for Conceptual Model validation. For completeness, validation rules are implemented by directly checking the gathered data for the Conceptual Model, e.g., a class must have name, one attribute being its identifier and one service. **Completeness** of the formal language specification of the Conceptual model, as checked by the validation process, means that: 1) there is no missing information in the formal specification detailing the requirements the code must meet; 2) stated in another way, all the required properties of the Conceptual Model encoded in the formal language specification are defined and they have a value. **Correctness** of the formal language specification of the Conceptual model, as checked by the validation process, means that: 1) when the statements in the formal language specification of the Conceptual model are both syntactically and semantically consistent and not ambiguous; 2) stated in another way, all the properties introduced in the conceptual model have a valid value. For correctness, an extended formal specification language grammar (syntax plus semantics) is implemented in order to validate the syntax and meaning of all the formulas in the Conceptual Model.

CORRECTNESS

More specifically, for completeness, the validtor functions to ensure that all the elements in a formal specification language have a set of properties that both exist and have a valid value. Most of the properties are strictly implemented to have a full definition and valid values.. Most of the properties are strictly implemented to have a full definition and valid values. However, the CASE tool 210 allows, for easy of use during a model inputting, to leave some properties incomplete or with invalid values. These properties will be checked by the validator 220 to be complete (and correct) prior to any automatic software production process.

The elements which are used to validate a Conceptual Model are described next. For each element it is stated if validation will be strict (e.g. when all his properties have to exist and must have a valid value at creation time) or flexible (e.g. validation will be accomplished at a later time). Some properties are optional, (e.g. that may not exist) but if they are defined, they must be validated. These elements are given in TABLE 5:

TABLE 5

- Class	
o Name.	Strict
o ID function	Flexible
o Attributes (at least one)	Flexible
o Services (at least Create service).	Flexible
o Static and Dynamic Integrity Constraints (optional)	
ß Their formula	Strict
- Attribute	
o Name.	Strict
o Type (Constant, Variable, Derived).	Strict
o Data-type (Real, integer, etc).	Strict
o Default Value.	Strict
o Size (if proceeds)	Strict
o Request in Creation service.	Strict
o Null value allowed.	Strict
o Evaluations (variable attributes).	Flexible
o Derivation formula (derived attributes).	Flexible
- Evaluation	
o One variable attribute of a class	Strict
o One service of the same class	Strict
o Condition (optional).	Strict
o Formula of evaluation.	Strict
- Derivation	
o Formula.	Strict
o Condition (optional).	Strict

- Service	
o Name.	Strict
o Arguments.	
ß argument's name	Strict
ß data-type	Strict
ß default value (optional)	Strict
ß null value	Strict
ß size (if proceeds)	Strict
o For a transaction, its formula.	Flexible
- Preconditions of an action	
o Formula.	Strict
ß Agents affected by condition	Strict
- Relationship: Aggregation	
o Related classes (component &composite)	Strict
o Relationship name.	Strict
o Both directions Role names.	Strict
o Cardinality.	Strict
o Inclusive or referential.	Strict
o Dynamic.	Strict
o Clause "Group By" (Optional).	Strict
o Insertion and deletion events (if proceed)	Strict
- Relationship: Inheritance	
o Related classes (parent & child)	Strict
o Temporal (versus permanent)	Strict
o Specialization condition or events	Strict

- Relationship: Agent	
o Agent class and service allowed to activate.	Strict
- State Transition Diagram (STD)	
o All states of class (3 at least).	Flexible
- State in STD	
o Name.	Strict
- Transition in STD	
o Estate of origin.	Strict
o Estate of destination.	Strict
o Service of class.	Strict
ß Control condition (optional).	Strict
- Trigger	
o Condition.	Strict
o Class or instance of destination.	Strict
o Target (self, object, class)	Strict
o Activated service.	Strict
o Service arguments' initialization (Optional)	
ß Arguments' values	Strict
- Global Interactions	
o Name.	Strict
o Formula.	Strict
- User exit functions	
o Name.	Strict
o Return data-type	Strict
o Arguments, (Optional)	

ß Argument's name	Strict
ß Argument's data-type	Strict

The table given in the above quoted passage indicates all the things that are checked during the correctness part of the validation process. The following passages from page 45 of the specification give meaning to what is entailed in the validation process in checking formal language statements for "completeness":

COMPLETENESS

Some properties of components in formal specification languages are "well formed formulas" that follow a well defined syntax. It is therefore, a requirement to ensure that all introduced formulas in the Conceptual Model were both syntactical and semantically correct.

Not all formulas used in the Conceptual Model have the same purpose. Therefore, there will be several types of formulas. Depending of formula's type, the use of certain operators and terms (operands, like: constants, class attributes, user-functions, etc.) are allowed. A process and a set of rules in grammar to validate every type of formula in the Conceptual Model also exists.

More specifically, the Conceptual Model includes formulas of the following types as shown in TABLE 6:

TABLE 6

- Default Value Calculation of
o Class Attributes (Constant and Variable)
o Service and Transaction Arguments
- Inheritance: Specialization condition

- Static and Dynamic Integrity Constraints

- Derivations and Valuations:

o Calculation formula (Derived or Variable attributes respectively)

o Conditions (optional)

- Preconditions for actions (Services or Transactions)

- Control Conditions for transitions in State Transitions Diagram

- Triggering conditions

- Local and Global Transactions formulas

These formulas are validated at the time they are introduced, by preventing the designer from leaving an interactive textual dialog if formula is not syntactically and semantically correct.

In general, every formula must be syntactically correct; every class must have an identification function; every class must have a creation event; every triggering formula must be semantically correct (e.g. self triggers to an unrelated class are forbidden); and every name of an aggregation must be unique in the conceptual schema. If these conditions are not satisfied, then an error is raised.

A warning may be raised, on the other hand, if any of the following do not hold: every class should have a destroy event; every derived attribute should have at least a derivation formula; every service should have an agent declared to execute it; and every argument declared in a service should be used.

Validation process will also be invoked every time the designer performs a change into the model that may invalidate one or more formulas. As mentioned earlier, for ease of use, certain type of formulas are allowed to be incorrect, which the designer will have to review at a later time. The automatic software production process in accordance with one embodiment of the present invention, however, will not continue to code generation, if not all the formulas are correct. Each time the designer introduces a modification in the Conceptual Model specification, all affected formulas will be checked. As a result, the following cases may happen:

1. If any of the affected formulas makes reference to a "Strict" property, the change will be rejected. An error will be raised to inform the designer.

2. If none of the affected formulas references a "Strict" property, modification to Conceptual Model will be accepted. An action-confirmation dialog is showed before any action is taken..

3. If there is no affected formula, modification is performed straightaway. In order to validate the user interface information, the validator 220 checks the following for errors: the patterns defined must be well constructed with no essential information lacking; the attributes used in filters must be visible from the definition class; the attributes used in order criteria must be visible from the definition class; the formula in a filter must be a well-formed formula using the terms defined in the model; the action selection pattern must use as final actions objects defined in the Conceptual Model; and the set of dependency patterns must be terminal and have confluence. Warnings may be generated under the following conditions: if a pattern is defined but not used (applied), if an instance pattern is duplicated .

Automatic software production from Conceptual Models requires these Conceptual Models to be correct and complete. Applying the characteristics and properties of formal specification languages makes it possible to effectively validate a Conceptual Model. The validation process is based on the grammar defined by the formal specification language, and partial validation is to be invoked any time the designer introduces modifications to the Conceptual Model specification. Prior to any automatic software production process, Conceptual Model will be validated in a full validation as a pre-requisite.

These passages give meaning to what the terms formal language and formal specification and validation for completeness and correctness in claim 1 and the other claims mean.

WHAT DOES THE PRIOR ART TEACH ABOUT PREPARATION OF FORMAL LANGUAGE SPECIFICATIONS FROM USER INPUT DEFINING A CONCEPTUAL MODEL

The above quoted passages from the specification help define the meaning of the claim terms "formal language" and "formal specification" and "automatically validating". The question then becomes does the prior art applied to claim 1 and the other claims teach preparation of formal language specifications. In particular, does the term "formal specification" in the article "Features of VDM tools" mean a specification for a complete, operable computer program written in a formal language, as called for by amended claim 1.

Goodwin et al. clearly do not teach use of a formal language specification since no formal language with a formal grammar is mentioned (the undersigned does not believe UML is a formal language) nor is it stated that a formal language specification is written. Goodwin et al. teaches at Col. 6, Lines 18 -22 that logical models 202 are input to the model adaptor which then outputs unified models 206 expressed in UML and that it is these unified models plus the system definition 208 which are provided as input to the code generator which generates source code of objects that can be used within the framework outlined by the developer.

Goodwin's data server feature does not do testing or validation of a formal language specification nor testing to ensure the conversion to unified model has been performed correctly because its purpose is different as can be seen from Co 15, Lines 49-52. "The data server 332 provides query services to access legacy databases, making available, if standard formats, data from a variety of sources." Goodwin teaches this as taking place at run time. Conversion takes place before run time, so any testing to ensure the conversion was done properly would have to occur before run time. The data servers therefore are not doing validataion. Their function is to allow queries to legacy databases. This is not validation as recited in the claims. <u>The Examiner seems to acknowledge this and points to the "Features of VDM tools" as teaching validation.</u>

The VDM Tools article does not teach automatic validation of both the syntax and semantics of a formal language specification.

First, consider the following evidence that the term "formal specification" as used in "Features of VDM tools" does not mean a formal language specification as used in the claims. **While the VDM Tools article does not specifically say the specification created by the VDM tool is not formal, there is adequate evidence that it is not a formal language specification in inference that can be drawn from the following statement. The "Features of VDM tools" article specifically states that the specification referred to is or can be "unstructured". If the VDM tools output specification can be unstructured, then it cannot be a formal language specification since all formal language specifications are highly structured with precise syntax and semantics.**

Further, the validations taught in VDM tools are limited to the Syntax Checker which only checks the syntax of the specification and the Type

Checker which only checks the modules/classes in a syntax checked file. This type checking is a static check for correctness of the use of the VDM concepts in the specification, I.e., whether the operands are of the right type for the operators with which they will be used and whether the variables used are in scope.

In summary, in VDM Tools, there is no automatic validation of the formal language specification which includes semantic checking as that term is used in claim 1 as intepreted from the teachings of the specification. The VDM Tools prior art reference does not fill the gap in knowledge in the prior art left by Goodwin et al.

KOOB VSE PRIOR ART AND WHAT IT TEACHES ABOUT VALIDATION

Nor does Koob fill the gap as evidenced by the following paragraph:

> "During formal development with VSE, a number of proof obligations arise that are automatically generated by the system and that can be carried out (proved) with the help of the available deduction subsystems KIV ...and INKA ....These obligations include, in addition to a proof of the security/safety properties of the specified system, requirement for ensuring the correctness of the implementation with regard to the specification."

This passage means what Koob teaches is the use of some automatically generated "proof obligations" as a set of "requirements for ensuring the correctness of the implementation with regard to the specification". Koob's proof obligations are things that must be done manually, or at most semi-automatically, to validate the software. There is no automatic validation of the software in Koob. This is proven by the following passage from the Koob Abstract:

"Beginning with a formal top level design in a formal specification language (VSE-SL), this specifcation can be transferred into executable programs through stepwise refinement. The proof obligations are automatically generated by the VSE system. The corresponding proofs can even be carried out semi-automatically....

Koob therefor does not teach a tool which automatically validates the specification to ensure it is "syntactically complete, semantically correct and not ambiguous" as required by claim 1.

WHAT DO THE TERMS SYNTAX AND SEMANTICS MEAN IN GOODWIN AND ARE THEY EVIDENCE THAT GOODWIN TEACHES USE OF A FORMAL LANGUAGE SPECIFICATION?

Another question is do references to "syntax" and "semantics" in Goodwin mean and are they the same meaning as these terms are used in claim 1. In Goodwin, the term "syntax" is associated to the syntax of templates. Col. 8, Lines 10-11. "The syntax of the templates supports any number of control structures followed by a block code." Col. 8, Lines 14-19. "This tailorability allows developers to implement their own template syntax (parser and interpreter) in which templates can be implemented. The preferred syntax for the templates is JavaScript, although any of a number of well know or custome syntaxes can be used with relative equal effectivity." Col. 15, Lines 2-5. "This tailorability allows developers to implement template syntax (parser and interpreter) in which templates can be implemented." Clearly these passages are evidence that syntax in Goodwin et al. means the syntax of templates for objects whereas in the application at bar syntax means the structure and order of elements of the formal language statements of the formal specification. The formal specification encodes the specific details of the user defined Conceptual Model which defines the computer program to be written, which is far different than a "cookie cutter" object template.

In Goodwin et al. , the term semantics is used to refer to a certain set of elements that compose an object model. Col. 5, Lines 28-30. "The attributes, inheritances, and relationships of all the object classes of an object model are called the 'semantics' or 'semantic elements' of the object model.

In contrast, in the application and claims at bar, "semantics" is the meaning of various parts of the formal language statements that make up the formal specification. A formal language must be used to practice the invention. The semantics of the formal language are the definitions of each element of a formal language statement. To be a formal language, everything must have one and only one meaning.

In summary, in Goodwin et al., semantics is used to define a set of elements that compose an object model. In the claimed invention, semantics means the definitions of the various parts of the formal language statements that comprise the formal specification.

The term "parsing" used in Goodwin et al. in the parsing portion of Figure 5 mean parsing of templates. Col. 13, Lines 52-56. "Thus, the code generator 330 can support the creation of, for example, IDL, Java or C++ files (CORBA, Java RMI, and/or COM) based on certain user preferences and selection. After user option is obtained (Block 508, Fig. 5), a template language parser is selected (Block 510, Fig. 5)."

Further, the only part of said templates that is parsed is the synax as Goodwin teaches at Col. 15, lines 2-5. "This tailorability allows developers to implement template syntax (parser and interpreter) in which templates can be implemented.

Summary of Non Obviousness Argument As to Claim 1

Given the above, it is apparent that there is no suggestion to combine the references since one skilled in the art would not perceive a liklihood of success of solving the problem the invention of claim 1 solved. This is because the combination would be missing several key elements needed to make the invention work. Specifically, none of the references teaches generation of a formal specification in a formal language along with automatic validation of both the syntax and semantics of the formal language statements in the specification.

Dependent Claims 2 - 6

<u>Claims 2-6 depend from claim 1, so they are not obvious since claim 1 is not obvious because there is no suggestion to combine the references.</u>

A few specific points. Claim 2 adds the notion that the formal specification is written by user input which defines graphically the elements of the conceptual model. Each graphical element is converted into one or more statements in the formal language specification.

Claim 3 adds the system logic translator. This feature is not taught by Koob et al. and is not related to the feature of claim 2 so it should not be rejected as obvious for the same reasons claim 2 was rejected.

Claim 4 adds a user interface translator which generates a GUI for the program defined in the conceptual model. The GUI referred to in claim 2 is the GUI of the CASE tool the user uses to graphically define the Conceptual Model of the program to be created. These are two different GUIs. This is part of the invention's ability in claim 4 to generate the software of a complete system including a user interface defined in the Conceptual Model. Koob does not teach a tool which can generate an entire system which includes a user interface. Koob teachs that a system can be divided into security critical and security uncritical system parts and teaches "conventional, I.e., manual, development of the security uncritical system parts. Specifically, Koob teaches,

> "On the specification level the interface is the EXPORT-specification of the formal development of the highest horiizontal level. The interfaces on the level the formal development ends are the EXPORT-specifications of the highest refinement level which are no longer realized through verified abstract programs; implementations are done through conventionally developed programs."

So again there is no suggestion because not all the elements needed to make the invention are present in the combined references.

Claim 5 adds a database generator. Koob makes no mention of the automatic production of a data structure or database structure.

Claim 6 adds a documention generator. No documentation generation is taught in Lee. The only reference to documentation found in Goodwin is Col. 6, Lines 44-45 where it is taught, "UML was first introduced in early 1997 and, like other modelling languages is well documented". The Pretty Printers feature of VDM tools teaches:

> "The formal specifcation is usually accompanied with supporting text and figures. The Prttty printer enables the production of such a mix by generating La Tex sources. The Pretty Printer can automatically produce cross-reference indexing of all definitions."

This is not a teaching of a tool that automatically generates documentation for a program from the formal specification thereof. The Pretty Printer is limited to mixing the formal specification as is with "supporting text and figures". It does not teach using the formal specification as an input to a documentation generator which then generates documentation for the program.

Independent Claim 7

The Examiner applied the same rejection logic to claim 7 as to claim 1, so the same counter argument detailed above for claim 1 applies to claim 7. All amendments were made voluntarily to improve the form of the claim and improve its accuracy.

Claims 8 - 12

Claims 8-12 are dependent from claim 7 and are not obvious for the same reasons claim 7 is not obvious and for the additional reasons claims 2-5 are not obvious. All amendments were made voluntarily to improve the form of the claim and improve its accuracy.

Claims 13-15

The Examiner rejected these claims for the same reasons he rejected claim 1. Accordingly, the arguments made for nonobviousness of claim 1 are incorporated by reference herein. All amendments were made voluntarily to improve the form of the claim and improve its accuracy.

Claims 17-23

The Examiner rejected these claims for the same reasons he rejected claim 1. Accordingly, the arguments made for nonobviousness of claim 1 are incorporated by reference herein. Claims 17 -23 all deal with the notion of valuations which is a formalism to define how the occurrence of an event changes the value of an attribute as part of the state of an object, and, more specifically, the means by which the user can graphically input the information necessary to define valuations, said information being later transformed into a formal language specification data structure. Koob does not teach such a valuation process or tools to allow valuation formulas to be input via a graphical user interface.

All amendments were made voluntarily to improve the form of the claim and improve its accuracy.

Claim 24

Claim 24 is an independent claim which is similar to claim 1 and which the Examiner has rejected on the same basis as claim 1. The arguments for non obviousness of claim 1 are hereby incorporated by reference.

All claims are now believed to be in condition for allowance.

Respectfully submitted,

Dated: June 25, 2004

Ronald C. Fish

Ronald Craig Fish
Reg. No. 28,843
Tel 408 778 3624
FAX 408 776 0426

I hereby certify that this correspondence is being deposited with the United States Postal Service as First Class Mail, postage prepaid, in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, Va. 22313-1450.
on 6/25/04
(Date of Deposit)

Ronald C. Fish

Ronald Craig Fish, President
Ronald Craig Fish, a Law Corporation
Reg. No. 28,843